Improving Test Suites for Efficient Fault Localization

Benoit Baudry, Franck Fleurey, Yves Le Traon

Improving Test Suites for Efficient Fault Localization

Benoit Baudry and Franck Fleurey
IRISA, Campus Universitaire de Beaulieu, 35042
Rennes Cedex, France
{ffleurey, bbaudry}@irisa.fr

Yves Le Traon
France Télécom R&D
2, av. Pierre Marzin - 22 307 Lannion Cedex – France
yves.letraon@francetelecom.com

ABSTRACT

The need for testing-for-diagnosis strategies has been identified for a long time, but the explicit link from testing to diagnosis (fault localization) is rare. Analyzing the type of information needed for efficient fault localization, we identify the attribute (called Dynamic Basic Block) that restricts the accuracy of a diagnosis algorithm. Based on this attribute, a test-for-diagnosis criterion is proposed and validated through rigorous case studies: it shows that a test suite can be improved to reach a high level of diagnosis accuracy. So, the dilemma between a reduced testing effort (with as few test cases as possible) and the diagnosis accuracy (that needs as much test cases as possible to get more information) is partly solved by selecting test cases that are dedicated to diagnosis.

Categories and Subject Descriptors

D.2.5 **[Software Engineering]** : Testing and Debugging.

General Terms

Measurement, Verification.

Keywords

Test generation, diagnosis, mutation analysis.

1. INTRODUCTION

In practice, no clear continuity exists between the testing task and the diagnosis one, defined in this paper as the task of locating faults in the program code. While the former aims at generating test data and oracles with a high fault-revealing power, the latter uses, when possible, all available symptoms (e.g. traces) coming from testing to locate and correct the detected faults. The richer the information coming from testing, the more precise the diagnosis may be. This need for testing-for-diagnosis strategies is mentioned in the literature [1, 9], but the explicit link from testing to diagnosis is rarely made. In [17], Zeller et al. propose the Delta Debugging Algorithm which aims at isolating the minimal subset of input sequences which causes the failure. Delta Debugging automatically determines why a computer program fails: the failure-inducing input is isolated but fault localization in the program code is not studied.

ICSE'06, May 20–28, 2006, Shanghai, China.

Considering the issue of fault localization, the usual assumption states that test cases satisfying a chosen test adequacy criterion are sufficient to perform diagnosis [1]. This assumption is verified neither by specific experiments nor by intuitive considerations.

Indeed, reducing the testing effort implies generating a minimal set of test cases (called a test suite in this paper) for reaching the given criterion. By contrast, an accurate diagnosis requires maximizing information coming from testing for a precise cross-checking and fault localization. For example, the good diagnosis results obtained in [9] are reached thanks to a large amount of input test data. These objectives thus seem contradictory because there is no technique to build test cases dedicated to an efficient use of diagnosis algorithms.

The work presented in this paper proposes a test criterion to improve diagnosis. This *test-for-diagnosis criterion* (TfD) evaluates the 'fault locating power' of test cases, i.e. the capacity of test cases to help the fault localization task. This TfD criterion allows bridging the gap between testing and diagnosis: an existing test suite which reveals faults is improved to satisfy the TfD criterion so that diagnosis algorithms are used efficiently. The goal is to obtain a better diagnosis using a minimal number of test cases.

To define the TfD criterion we identify the main concept that reduces the diagnosis analysis effort. It is called Dynamic Basic Block (DBB) and depends both on the test data (traces) and on the software control structure. The relationship between this concept and the diagnosis efficiency is experimentally validated. Experimental results also validate the optimization of test suites that satisfy the TfD criterion, in comparison with coverage-based criteria.

All the experiments use the algorithm proposed by Jones et al. [9] for diagnosis. We apply a computational intelligence algorithm (bacteriologic algorithm [3]) to automatically optimize a test suite for diagnosis, with respect to a criterion. Finally, we use mutation analysis [6, 14] to systematically introduce faults in programs. The efficiency of a test suite for fault localization is estimated on the seeded faults. This estimate experimentally validates the benefit provided by the TfD criterion based on DBB for fault localization.

Since the scalability issue is crucial when dealing with fault localization, the whole approach is integrated in an optimization process which allows dealing with the possibly large size of the program under diagnosis.

Section 2 details the algorithm proposed by Jones et al. in [9]. Section 3 investigates the relationship between testing and diagnosis. The proposed model identifies a test criterion that fits

the diagnosis requirements. Section 4 details a technique to automatically generate test cases with respect to the criterion defined in section 3. Section 5 presents the experimental validation of the technique while section 6 discusses the practical use and the scalability of the technique in the testing/debugging process of a program. Section 7 concludes this paper.

2. BACKGROUND ON DIAGNOSIS ALGORITHMS

After the failure of some test cases on a program, the debugging process consists, first in locating the faults in the source code (this is called *diagnosis*), and, second, in fixing them. To reduce the cost of diagnosis several techniques are presented in the literature to help the programmer locate faults in the program code. Those techniques mainly consist in selecting a reduced set of "suspicious" statements the tester should examine first to find faults.

2.1 Cross checking strategies and diagnosis accuracy

The cross-checking diagnosis algorithms correlate the execution traces of test cases, using a *diagnosis matrix* as presented in the left part of Figure 1. The matrix represents the execution traces for a set of test cases and the associated verdicts. Based on this matrix, the algorithms determine a reduced set of "suspicious" statements that are more likely to be faulty.

As an example, Figure 1 presents the code of a function that computes the power y of x. A fault has been introduced in the algorithm at statement {3} (the correct statement would be **p:=-y**) and, a diagnosis matrix is presented for four test cases. Test case 3 detects the fault. Based only on this test case, the 4 statements executed by test case 3 are equally suspected.

Cross-checking diagnosis strategies correlate several test case executions to order the statements from the less to the most suspect. Considering the 4 test cases and statement 4, one may notice that it is not executed by the failed test case and executed twice by passed test cases. Intuitively, this statement appears as less suspect than the others. The cross-checking strategies differ from one another by the way they correlate test cases traces to locate faults.

The relevance of the results of a diagnosis algorithm can be estimated by the number of statements one has to examine before finding a fault. We define the *diagnosis accuracy* as the number of statements to be examined before finding the actual faulty statement and the *relative diagnosis accuracy* as the corresponding percentage of the source code of the program to examine.

> **Diagnosis accuracy.** *For an execution of a diagnosis algorithm, the diagnosis accuracy is defined as the number of statements one has to examine before finding a fault.*

Example: In Figure 1, the diagnosis accuracy obtained with the only test case 3 is 4 since 4 statements are equally suspected (≈ 57%).

2.2 Existing cross-checking techniques

This section introduces several cross-checking algorithms from the literature.

In [1], Agrawal et al. propose to compute, for each test case, the set of statements it executes (dynamic slice) and then to compute the differences (or dices) between the slices of failed and passed test cases. The intuition is that the faulty statement should be in those dices. But, as the number of dices to examine may be important, the authors propose, as a heuristic, to examine dices from the smallest to the biggest. In this context, the authors present a tool called XSlice to display dices by highlighting the suspicious statements in the component's code. The approach is validated on a C program (914 lines of code) by injecting up to 7 bugs at the time and using 46 test cases generated by a static test data generation tool.

In [10], Khalil et al. propose an adaptive method to reduce the set of suspicious statements. First, assuming that only one statement is faulty and that verdicts are "ideal", the algorithm cross-checks the positive (which verdict is pass) and negative (verdict fail) execution traces to pinpoint the suspect statements. The authors then describe an adaptive strategy which incrementally releases the first "single fault" and "ideal verdicts" assumptions, until finding the actual faulty statement. The approach is validated by injecting faults in several VHDL and Pascal small programs.

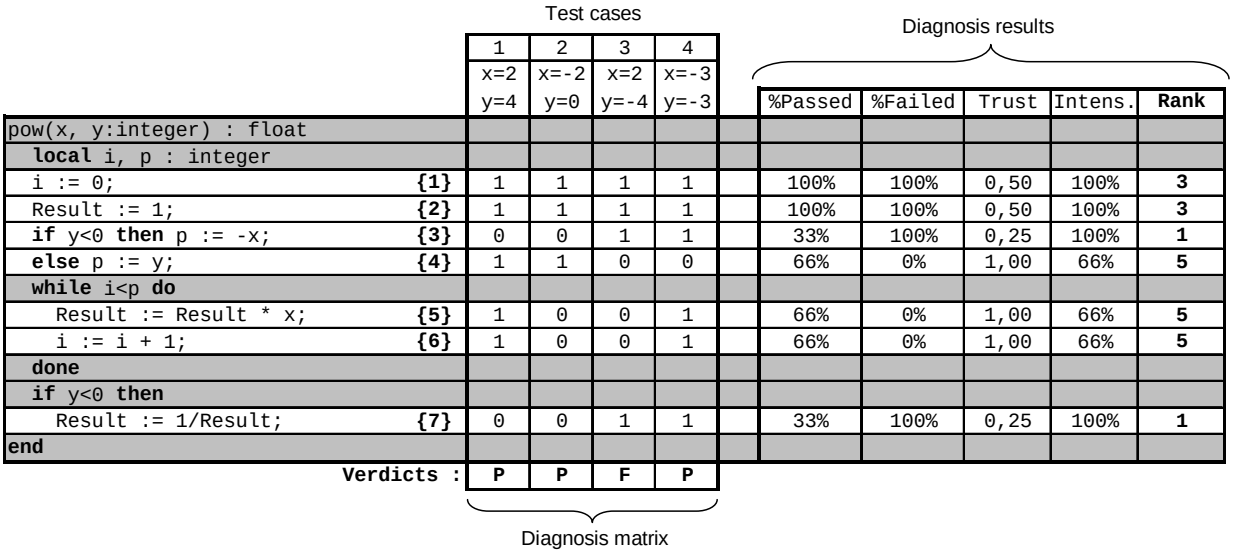
In [5], Dallmeier et al. establish a ranking of the suspicious classes in a Java program by analyzing incoming/outgoing sequences of class method calls (which are called *traces* in that context). The mathematical model strongly depends on the "distance" between passing and failing runs. The model highlights the classes which behave very differently between passing and failing runs. A deviation is relevant in terms of diagnosis iff the program runs are strongly related. In their case study, with an average number of 10.56 executed classes, over 386 program runs, the algorithm reduces the search to 2.22 classes while a random placing of the faulty class would result in an average search length of 4.78 classes. Their conclusions are also relative to this sole experiment. In [4], the authors introduce the notion of cause transition to locate the software defect that causes a given failure.

The Tarantula approach proposed by Jones et al. [9] makes few assumptions on the quality of verdicts and on the number of faulty statements. It is validated experimentally with up to 7 faults to locate at the same time. In [8], an empirical study validates Tarantula as the best existing technique for fault localization. Thus, we have chosen it for our experiments, and the following presents more details.

The idea of the algorithm is that faulty statements more frequently appear in the traces of failed test cases than in passed test cases. The algorithm thus orders statements according to a *trust value* computed from the diagnosis matrix (right part of Figure 1). This corresponds to the ratio between the percentage of passed test cases that execute a given statement and the total percentage of test cases that execute this statement.

In addition to this measure, another value is computed for each statement. This value, which we call *Intensity(s)* for a statement s, corresponds to the maximum between the percentage of passed test cases and the percentage of failed test cases that execute this statement. The intuition is that the higher this value is the most accurate the trust measurement should be.

Let us notice that in [9], Jones et al. propose a tool to visualize the results of diagnosis, the notions of trust and intensity are thus called colour and brightness. Since we do not use explicit visualization here, we find it more appropriate to propose a new vocabulary not based on visual ideas.



	1	2	3	4		%Passed	%Failed	Trust	Intens.	Rank
	x=2 y=4	x=-2 y=0	x=2 y=-4	x=-3 y=-3						
pow(x, y:integer) : float										
local i, p : integer										
i := 0; **{1}**	1	1	1	1		100%	100%	0,50	100%	**3**
Result := 1; **{2}**	1	1	1	1		100%	100%	0,50	100%	**3**
if y<0 **then** p := -x; **{3}**	0	0	1	1		33%	100%	0,25	100%	**1**
else p := y; **{4}**	1	1	0	0		66%	0%	1,00	66%	**5**
while i<p **do**										
Result := Result * x; **{5}**	1	0	0	1		66%	0%	1,00	66%	**5**
i := i + 1; **{6}**	1	0	0	1		66%	0%	1,00	66%	**5**
done										
if y<0 **then**										
Result := 1/Result; **{7}**	0	0	1	1		33%	100%	0,25	100%	**1**
end										
Verdicts :	**P**	**P**	**F**	**P**						

Figure 1 – Diagnosis matrix and results

Trust value and intensity of a statement. *Let s be a statement, %Passed(s) the ratio of passed test cases that execute s and %Failed(s) the ratio of failed test cases that execute s. The trust in the statement s is computed as:*

$$Trust(s) = \frac{\%Passed(s)}{\%Passed(s) + \%Failed(s)}$$

$$Intensity(s) = Max(\%Passed(s), \%Failed(s))$$

The technique orders all the statements of the program using the *Trust* value as the major component and the *Intensity* value as the tie-breaker. Statements are then manually examined following the computed order to find the actual faulty statement. For example, Figure 1 shows the *Trust* and *Intensity* values, as well as the *Rank* for each statement of the `pow` function. The statement ranked 1 is the most suspect statement. In this particular example, it happens to be the actual faulty statement. The statements that have the same ranking are given a rank that corresponds to the lowest number of statements that would need to be examined if one of these statements was the faulty one.

3. FROM TEST TO DIAGNOSIS

This section presents the ideas and discussion that lead to the main contribution of this work: a test-for-diagnosis criterion. As the diagnosis uses information collected during the test, the intuition is that the diagnosis should be as accurate as the number of test cases is high. Unfortunately, this idea is contradictory with test generation practices which consist in minimizing the number of test cases to satisfy a given test criterion. To deal with this we discuss several test criteria in order to fit the diagnosis requirements.

3.1 Code coverage based criteria

In order to detect and locate faults anywhere in the program, any diagnosis technique requires each statement to be covered by the tests. So, a first test criterion for an accurate diagnosis is *statement coverage*. However, this criterion is inadequate for diagnosis. For example, using Jones et al. algorithm, if the test cases simply cover the code, the ranking produced by the algorithm may contain many *indistinguishable statements* (same values for trust and intensity). This produces poor results for diagnosis as the actual faulty statements may be lost in a large amount of indistinguishable correct statements.

This analysis of diagnosis algorithms leads us to define a second test criterion for diagnosis: having least N test cases that cover each statement of the program. In the following we call this criterion *N-Coverage* (1-Coverage corresponds to simple statement coverage). The intuition is that the higher is N, the more statements can be distinguished by the test cases.

However none of these criteria focuses on the specificity needed for good fault localization: the algorithm needs test cases that enable it to distinguish statements one from the other. Next section introduces an original test-for-diagnosis criterion directly based on statements distinction, which specifically aims at improving the diagnosis.

3.2 Distinguishing statements

The diagnosis objective being to pinpoint faulty statements in a program, it requires being able to distinguish any statement of the program from each others. In other words, no matter how sophisticated the technique is, if its inputs do not allow distinguishing statements (particularly the faulty ones from the others), it will fail producing an accurate diagnosis. In this paper, as we focus on diagnosis techniques based on cross-checking test execution traces, the particular input we have to deal with is the diagnosis matrix. Thus, to improve diagnosis accuracy the test suite must be designed to minimize the number of indistinguishable statements in the diagnosis matrix.

3.2.1 Dynamic basic blocks

If two statements have identical lines in the coverage matrix, they are indistinguishable. On the matrix presented Figure 1, for instance, the statements {3} and {7} covered by test cases 3 and 4 are indistinguishable, independently of the diagnosis algorithm

which may be used. From this observation, we introduce the notion of *dynamic basic block* (we chose the name as a reference to basic block defined for compilers: any basic block is included in a DBB). In practice, dynamic basic blocks (DBB) can be easily computed from the diagnosis matrix by grouping statements that are covered by the same test cases. For example, we can identify four dynamic basic blocks in the coverage matrix of Figure 1: {(1, 2), (3, 7), (4), (5, 6)}.

Dynamic basic block. *Let P be the program under test and TS a test suite (a set of test cases). A dynamic basic block DBB is the set of statements of P that is covered by the same test cases of TS. Two statements s and s' belong to a DBB if they have identical lines in the coverage matrix.*
The set of dynamic basic blocks in P, distinguished by TS, is denoted B(TS).

3.2.2 Size of the DBB and diagnosis accuracy

This section investigates the relationship between distinguished DBBs and the diagnosis accuracy. The models and discussion presented here are relevant for any diagnosis algorithm that uses the diagnosis matrix as an input.

By the definition of the DBB itself, all statements in a single DBB are indistinguishable. This way, if one statement is classified as suspect by the diagnosis algorithm, then every statement that belongs to the same DBB will also be selected as suspicious. Using Jones et al. algorithm for instance, every statement from a single DBB has the same values for Trust and Intensity and comes out with the same Rank. We can interpret the results of the diagnosis algorithm as a ranking of the DBBs distinguished in the program.

Based on this interpretation, let us define the theoretical notion of *optimal diagnosis algorithm.* This "virtual" algorithm always selects the dynamic basic block containing the actual faulty statement as the most suspicious.

With this notion of optimal diagnosis algorithm, one can model the ideal accuracy of diagnosis under the following assumptions:

1 Faults are uniformly distributed: each statement has the same probability to be faulty.

2 The diagnosis algorithm is optimal.

Ideal accuracy of diagnosis. *Let P be the program under test, TS the test suite and B(TS) the set of dynamic basic blocks distinguished by TS in P (as defined above). The average amount of suspected code is:*

$$Accuracy(TS, P) = \sum_{b \in B} p(b) \times n(b) = \sum_{b \in B} \frac{|b|^2}{2 \times |P|}$$

where:

p(b) is the probability that the fault is located in $b \in B(TS)$. We have $p(b)=|b|/|P|$.

n(b) is the average amount of code to examine. We have $n(b)=|b|/2$.

This estimate, based on an ideal diagnosis, reveals the main parameter which impacts on diagnosis accuracy: the size of a dynamic basic block. In fact, for a particular program, the ideal diagnosis accuracy only depends on the sizes of the DBB (in section 5.2 the experiments show that the size of DBBs is also decisive even if the algorithm is not ideal). To improve the accuracy of the diagnosis, the size of dynamic basic blocks should be minimal.

We can notice that the actual minimal value for the size of the DBB is the size of the smallest block of statements sequentially executed. This minimal size is generally not statically decidable. So, the TfD criterion to improve the accuracy of diagnosis selects a test suite that maximizes the number of dynamic basic blocks. The maximum possible number is the number of static basic blocks. However, static blocks can be indistinguishable, and it is not possible to statically decide if they are. A better target for the maximum number of DBBs is the number of control-dependence regions, because we know that basic blocks in such regions are indistinguishable. However, it is still possible that two regions are indistinguishable, and again, it is not possible to statically know that.

3.3 Test for diagnosis criterion

From the discussions presented earlier, we can conclude that a test suite must at least cover the code to be useful for diagnosis, and, if it covers the code N times diagnosis is improved. This N-coverage (N > 0) criterion is a minimum requirement to apply the diagnosis algorithm. Going further in the analysis of the problem we have defined an original test criterion that is dedicated to diagnosis. This criterion is called a test-for-diagnosis (TfD) criterion. In the following of the paper, we investigate and compare the relevance, for fault localization, of the three criteria proposed in this section: *Coverage*, *N-Coverage* and *TfD*.

Test-for-Diagnosis (TfD) criterion. *A test suite satisfies the TfD criterion if it maximizes the number of dynamic basic blocks distinguished in the program under test.*

The TfD criterion is different from classical test criteria in two ways. First, it has a different role: the expected role of test adequacy criteria is to qualify the fault-revealing power of the tests while, the TfD criterion qualifies the ability of a test suite to optimize fault localization. Second, this criterion guides the test generation (while the generation distinguishes new DBBs, it continues), but it does not provide an exact stopping criterion (in terms of an exact number of elements that should be covered).

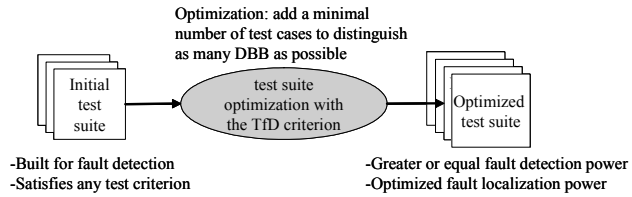


Figure 2 -Global test suite optimization process

In practice, the TfD criterion is used to optimize a test suite that has been generated with other criteria and that detects faults in a program. The idea is to use this criterion to improve the "fault localization power" of an existing test suite. Figure 2 summarizes this particular usage. Next section presents the bacteriologic algorithm that can be used to automatically optimize a test suite.

4. AUTOMATIC TEST OPTIMIZATION

In this section, we discuss the problem of automatically optimizing test cases that satisfy the criteria defined in the previous section. This section adapts the algorithm proposed in [2,

3] to optimize a test suite. As the experimental studies in section 5 will show, it is well suited for the three criteria we are studying in this paper.

4.1 Dynamic test data generation

Dynamic test data generation mainly consists in a function that associates a fitness value (related to a test criterion) to each input of the system. The value of this function is dynamically computed during the system's execution. The idea is to use this feedback information to search test data that satisfy the considered test criterion. In practice, test data are incrementally modified to optimize their fitness value. In this context, the test optimization problem becomes an optimization problem. Traditional optimization algorithms like gradient descent have been applied to solve this problem [11], but, the most efficient techniques are based on genetic algorithms that have been successfully applied in several works [13, 16]. However, these works focus on generating one test case for each test objective but cannot be applied to generate a global test suite. This is the reason why our approach focuses on the global optimization of a test suite with a dedicated algorithm, called bacteriologic algorithm.

4.2 The bacteriologic approach

The bacteriologic algorithm is an original adaptation of genetic algorithms as described in [3]. It is designed to automatically improve the quality of a test suite. The aim of this algorithm is to generate an efficient test suite for a given component under test. The algorithm also takes into account the number of test cases in the generated set. Since it is specialized for test cases generation, it is more efficient than the genetic algorithm (faster convergence, easier to tune).

The algorithm is a pseudo-random algorithm based on the biological process of the bacteriologic adaptation [15], and aims at generating a test suite that satisfies a given criterion. The algorithm takes an initial test suite as an input (each test case being modelled as a *bacterium*). Its evolution consists in series of mutations on bacteria, to explore the whole scope of solutions. The final test suite is incrementally built by adding bacteria that can improve the quality of the set. Along the execution there are thus two sets, the solution set that is being built, and the set of potential bacteria.

Bacterium modeling. *A bacterium is a test case. In the special case of system testing, a bacterium is an ordered set of commands. This set must be a legal input for the system under test.*

Two operators are needed for this algorithm: a *bacterium mutation operator*, and a *fitness function* to evaluate the quality of a given set of bacteria. The bacterium mutation operator consists in slightly altering the value of bacteria to create a new one that carries other information. For the case studies presented in this paper, it replaces a command in the set by another licit command.

Bacterium mutation operator. *Let $B=[c_1,\ldots, c_n]$ be a bacterium composed of n commands. Let c_i be a randomly selected command in B. The bacterium mutation operator consists in replacing c_i by a randomly generated valid command c'_i.*

$$B=[c_1,\ldots, c_i ,\ldots, c_n] \rightarrow B=[c_1,\ldots, c'_i ,\ldots, c_n]$$

The fitness function computes the quality (*fitness value*) of a set of bacteria for a particular criterion. This function serves two purposes: stop the algorithm when the fitness value of the solution set reaches a particular value, and evaluate the information a bacterium can add to the solution. Along the execution of the algorithm, a bacterium is added to the solution set if it can improve the quality of the set. The quality of a bacterium at a given moment is evaluated by the fitness value the solution set would have if this bacterium was added. We define a fitness function for a bacterium as follows.

Fitness function for a bacterium. *Let S be a set of bacteria, and F a fitness function. The fitness f(b) of a bacterium b is computed as follows: $f(b) = F(S\cup\{b\})-F(S)$. The more information the bacterium can bring to improve the set, the greater fitness value it has.*

The fitness value for a test case, is thus related to its efficiency to satisfy a given test criterion. Based on the criteria identified in the previous section, two fitness functions are defined in the following to optimize test cases for an efficient diagnosis.

Fitness function for statement coverage. *Let S be a test suite for a program P, $|P|$ the number of statements of P, and $|C(S)|$ the number of statements of P covered by S. The fitness function F for statement coverage for S is defined as:* $F(S) = \frac{|C(S)|}{|P|}$.

Fitness function for TfD. *Let S be a test suite for a program P and $|B(S)|$ the number of dynamic basic blocks distinguished by S in P, the fitness function F for TfD is defined as: $F(S) = |B(S)|$.*

These two fitness functions are based on the statement coverage and TfD criteria. A test suite that satisfies the N-Coverage criterion can also be generated using a bacteriologic algorithm. Since the bacteriologic algorithm is a pseudo-random algorithm, the resulting test suites are not the same from one execution to another. It is thus possible to produce N tests suites that each covers all the statements and to merge them to get a new test suite which cover at least N times each statement.

5. EXPERIMENTAL VALIDATION

The experiments conducted with two case studies aim at validating the ideal model of diagnosis presented section 3.2 and at comparing the relevance of the TfD criterion with N-coverage. The section starts with the presentation of the experimental process and the tools we used. Then, it presents the systems under test we studied and details the obtained results.

5.1 Experimental process and tools

The experimental process used to validate the approach is presented in Figure 3. It consists of 5 steps:

1. The initial test suite is optimized using a bacteriologic algorithm and based on the chosen fitness function (e.g. TfD criterion).
2. Mutants are generated for the program under test
3. Test cases are executed against all mutants. Verdicts and execution traces are collected.
4. Based on the results collected at previous step, a diagnosis matrix is built for each mutant
5. The diagnosis algorithm is executed for each mutant

The whole process can be automated using specific tools. JTracor, produces the execution trace for a particular execution of a program. JMutator, is a mutation tool for Java which produces

mutants for a program using 7 mutation operators detailed in Table 1, and runs a test suite on each mutant. Mutation analysis allows the generation of many faulty versions of a program, and thus provides trends and replicable results. The source code and documentation for these tools are available at [7]. The bacteriologic and Jones' fault localization algorithms have also been implemented for Java programs.

Table 1 - Mutation operators implemented by JMutator

Mutation operator	Abbreviation
Additive operator Insertion	AI
Constant Replacement	CR
Identifier by Constant Replacement	ICR
Identifier Replacement	IR
Relational Operator Replacement	ROR
Statement Deletion	SD
Unary Operator Insertion	UOI

In the first step of the experimental process (Figure 3), the initial test suite that covers all statements is generated. Then, it is optimized with the bacteriologic algorithm to satisfy the N-Coverage and TfD criteria. It has to be noticed that no oracle function is needed for the test suite optimization: a new test case is produced based on the fitness function, which is automatically computed with the execution trace. At the end of the process, an oracle is required only for the additional test cases which have been selected to satisfy the TfD criterion.

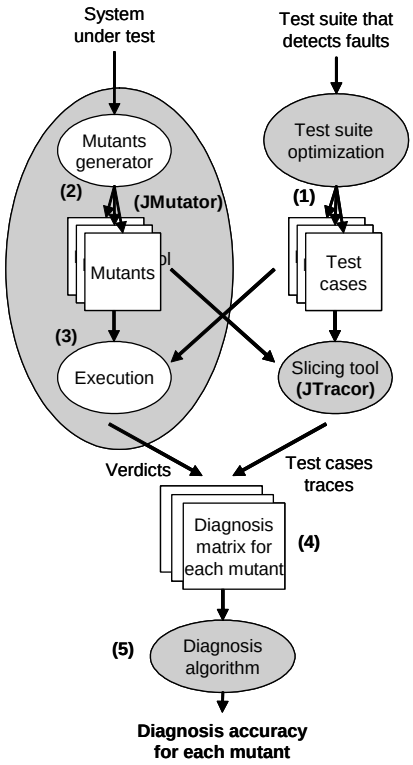


Figure 3 – Experimental process

Once a test suite is obtained, the experiment aims at estimating its quality for fault localization. This consists in executing the test suite on several faulty versions of the system under test (mutants), and then applying the fault localization algorithm to evaluate the accuracy of diagnosis.

JMutator executes the test suites on each mutant and records the verdicts for each test case on each mutant. JMutator uses the usual oracle for mutation: a test case fails iff the result of the mutant is different from the result of the correct program. Thus, for the case studies, the oracle function is automatic. The test cases are also executed with JTracor to obtain their execution traces. Both execution traces and verdicts are then used to compute the diagnosis matrix for each mutant version of the program.

The localization algorithm is applied, for each mutant, with the diagnosis matrix, to order the program's statements from the most to the least suspicious. Since the actual faulty statement is known by the tool, for each mutant, it is possible to determine the position of this statement in the list produced by the diagnosis algorithm (diagnosis accuracy). The closer it is from the beginning of the list the more accurate the diagnosis is. Using all mutants, the average diagnosis accuracy is computed, it estimates the quality of the test suite for the diagnosis.

5.2 Systems under test

To study the relevance and compare the criteria proposed for diagnosis we apply the experimental process previously described with two object-oriented systems. The first one, BOOK is a small example used to study the relationship between the diagnosis accuracy and the DBB sizes. It consists in a library management sub system composed of 16 classes for a total of 247 executable lines of codes (excluding comments and blank lines). The second system, VIRTUALMEETING (VM), is a server that simulates business meetings over network. It is made of 72 classes and 1478 executable lines of code. We validate the TfD criterion on this larger system. The source code of both systems can be downloaded from [7]. We believe that applying the diagnosis process on 346 faulty programs is sufficient to obtain confidence in the TfD criterion.

The inputs of these systems are textual commands, thus a test case for such systems is a sequence of syntactically correct commands.

5.3 Diagnosis accuracy vs. DBB size

The first study, with the BOOK system, validates the relationship between the diagnosis accuracy and the size of the DBB that contains the faulty statement.

5.3.1 Test suite optimization

Following the experimental protocol, first tests for the BOOK system are generated. With a fitness function based on code coverage, the bacteriologic algorithm optimized several test suites. Six test suites were obtained, each composed of 7 to 9 test cases and covering over 95,6 % and 96,4 % of the system's code. Let us notice that the code not covered by the test cases appears to be some exception handling code that is not reachable in the context of our experiments (it consists for instance in catching Input/Output errors when reading the input file).

By merging these test suites, we obtain a test suite composed of 47 test cases that covers 96.4% of the code. This test suite satisfies the 6-coverage criterion and is used to experimentally investigate the relationship between diagnosis accuracy and DBB size.

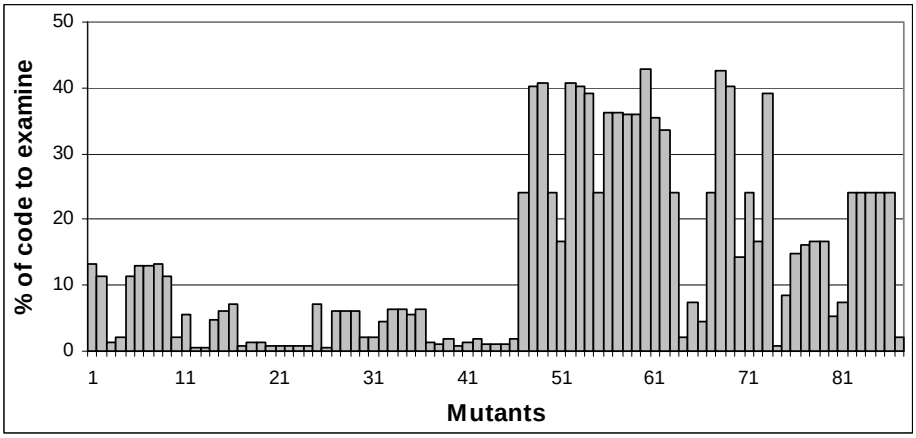


Figure 4 - ratio of suspicious code for each mutant

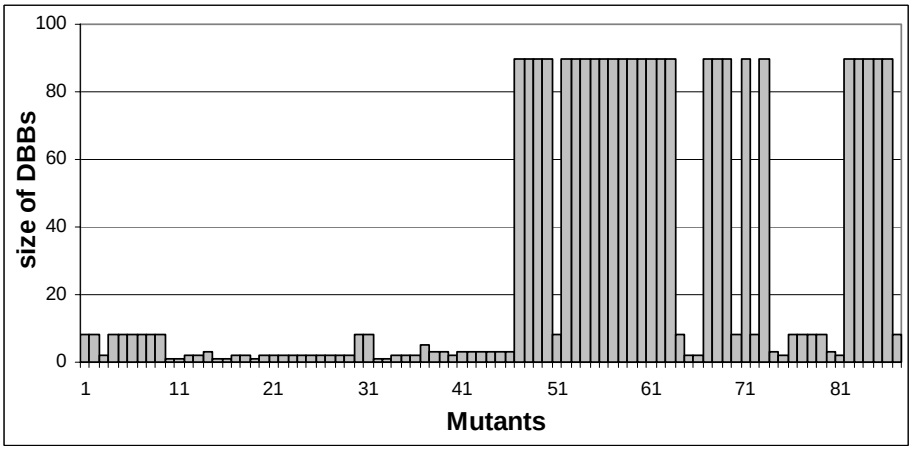


Figure 5 - Sizes of the DBBs containing the fault

5.3.2 Faulty versions of the system

Using JMutator, we obtained 96 mutants of the BOOK system. Each mutant contains a single fault located in any class of the system. For information, we looked at the quality of the test suite in terms of fault detection: the proportion of mutants detected by the test cases, also called *mutation score*. This score is computed by the mutation tool and is equal to 90.6% for the test suite (87 mutants out of 96 are detected). Among the 9 undetected mutants, 4 were not detected because the faulty code is not covered (exception handling code), and, 5 because some behaviour is not tested.

5.3.3 Results

Jones' diagnosis algorithm is applied on the 87 mutants detected by the test suite (the diagnosis algorithm cannot help locating a fault if no test case detects it).

The results displayed on Figure 4 clearly illustrate two different behaviours for the algorithm: faults in mutants 1 to 46 are easily located (in average 4,3% of the system's code has to be examined before locating the fault) while faults in mutants 47 to 87 are very difficult to locate. For these last mutants an average of 24.76% of the system code is suspected and it can go up to 43% for mutants 60 and 68.

Using the diagnosis matrices, we computed the size of the dynamic basic block that contains the actual faulty statement for each mutant (Figure 5). For mutants 1 to 46 the blocks have an average size of 3.4 whereas the size of DBBs for mutants 47 to 87 is mostly 90 with few small blocks of 8 statements (average is 60).

It appears that the faults that are difficult to localize are mostly located in large dynamic basic blocks (comparing Figure 4 and Figure 5), which shows a strong correlation between the size of the dynamic basic block and the accuracy of the diagnosis algorithm. For this particular experiment, the large dynamic basic block corresponds to the initialization code of the BOOK system.

5.3.4 Conclusion and discussion

The mathematical model of an ideal diagnosis algorithm presented section 3.2 identified the size of the dynamic basic blocks as decisive for diagnosis accuracy. The results obtained on the BOOK system confirm a strong correlation between the size of the DBB and the diagnosis accuracy: the algorithm performs much better when the DBB containing the fault is small. Similar results have been observed with the virtual meeting case study.

This study also shows that the diagnosis algorithm we use is not ideal: the diagnosis accuracy varies from one mutant to another even if the faulty statement is in the same DBB. This means that the actual faulty DBB is not always ranked as the most suspicious

according to this algorithm. In practice, this is mainly due to the problem of ideal verdicts: some test cases pass even if they execute a faulty statement.

This experiment illustrates that, although the diagnosis algorithm is not ideal, the notion of dynamic basic blocks is relevant regarding the diagnosis accuracy. This result is encouraging since the TfD criterion we propose section 3.2 is based on this idea.

5.4 Minimizing DBB to improve diagnosis

In this section, we experimentally compare the TfD criterion with N-coverage using the VIRTUALMEETING case study. The aim of this study is to estimate the benefit provided by using a TfD criterion based on minimizing DBB rather than coverage criteria to optimize a test suite for diagnosis.

5.4.1 Test suite optimization

First, test suites are produced. For Coverage and N-coverage criteria we use a bacteriologic algorithm with a fitness function based on code coverage. For the TfD criterion we also use a bacteriologic algorithm to optimize a test suite with the fitness function based on the number of distinguished DBB presented in section 4.2.

As in the previous study, several test suites are built, each of them covering the code of the system. Then, by merging N of those test suites we built test suites that satisfies the N-Coverage criterion. Running the bacteriologic algorithm 4 times, with an initial test suite randomly generated, we obtained 4 test suites composed of 14 to 18 test cases, each covering around 87% of the code of the application. In [7], we study these results in detail and show that only 88.8% of the code is coverable (the rest is dead code and exceptional code). It has to be noticed that the goal of this study is to enhance an existing test suite which already detects faults. Even if the whole code is not covered, and even if the test suite does not fully satisfy a chosen test adequacy criterion, the test suite can be optimized for locating the detected faults. This illustrates the difference between a test adequacy criterion and the TfD one. Mutants injected in the non-covered code were not considered in the study, since faults were not detectable by the test cases.

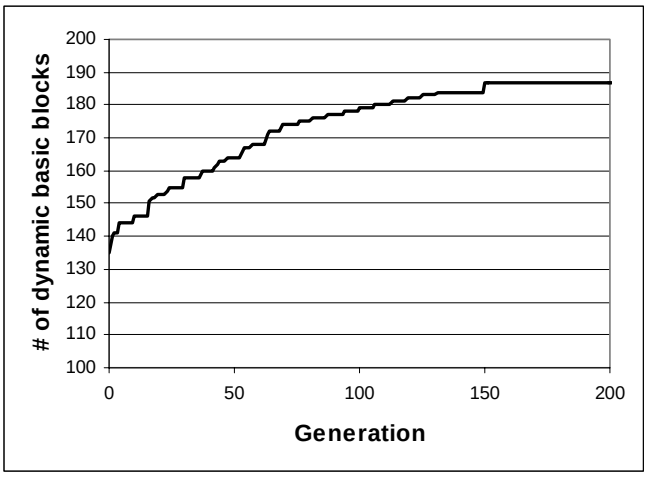


Figure 6 – Bacteriological algorithm maximizing the number of DBBs

To reach the TfD criterion, from one initial test suite that covers all statements (15 test cases), the algorithm adds a test case to the test suite if it allows distinguishing more DBBs in the program under test than the test suite already selected.

Figure 6 presents an execution of the bacteriologic algorithm. It shows the growth of the number of DBBs distinguished along the execution of the algorithm. As initial test suite, we use a test suite previously generated to cover the code of the application composed of 15 test cases that distinguish 113 DBBs. After around 150 iterations of the algorithm the final test suite is composed of 31 test cases and is able to distinguish 186 DBBs.

5.4.2 Faulty versions of the VM system

Using JMutator, 250 mutants were generated by injecting faults in the functional code of the system. No faults were introduced in the dead and exceptional code as they cannot be detected by the generated tests. We also chose not to introduce faults in the initialization code as the diagnosis algorithm is not adequate for this code executed by every test case. This problem was first identified by Jones et al. in [9] and confirmed by the study on the BOOK system presented previously.

5.4.3 Results and discussions

Table 2 – Test suites for the virtual meeting

Test suite	Code coverage	Number of DBBs
TS1	86,60	113
TS2	86,94	148
TS3	88,56	177
TS4	88,56	182
TS5	86,94	186

Table 2 summarizes data for the test cases optimized for the virtual meeting. TS1 to TS4 have been obtained using criteria from 1-Coverage to 4-Coverage and TS5 with the TfD criterion. Figure 7 displays the number of test cases in each test suite and the diagnosis accuracy obtained using each test suite. The accuracy is the average number of statements that must be analyzed to locate the fault. It is computed for the 250 mutants of the VM system.

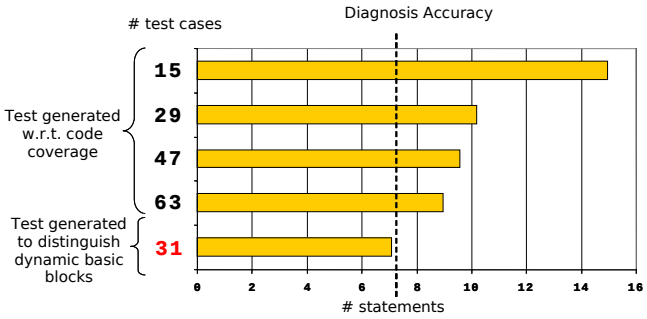


Figure 7 - Results summary for the virtual meeting system

This study confirms the intuition that using an N-Coverage criterion, the number of distinguished DBBs increases and the diagnosis accuracy is better when N increases. With the 1-Coverage criterion, an average of 15 statements has to be examined to find the actual faulty statement while using 4-Coverage only 9 statements have to be examined. Yet, increasing N also significantly increases the number of test cases: from 15 test cases for 1-Coverage to 63 test cases for 4-Coverage. Studying the diagnosis accuracy of the optimized test suite with TfD criterion shows that the average number of statements to

inspect is divided by two with respect to the initial test suite (from 14.95 to 7.09 statements).

The TfD criterion seems to fit better the diagnosis requirements: with only 31 test cases, the test suite TS5 optimized with the bacteriologic algorithm is able to distinguish 186 DBBs and leads to an average diagnosis accuracy of 7 statements. In fact, with half as many test cases as in TS4 this test suite allows a better diagnosis accuracy.

To conclude, this study confirms the intimate relationship between the notion of DBB and the accuracy of diagnosis. It experimentally validates the benefit provided by the TfD criterion for fault localization: it allows a better diagnosis using a small number of test cases.

Next section discusses the practical use of this criterion in the test/debugging process.

6. THE *TEST-FOR-DIAGNOSIS* PROCESS

We now study how to apply the proposed approach in a real-diagnosis process, through an incremental methodology to deal with large scale programs.

6.1 Methodology

The diagnosis aid techniques considered in this paper are helpful for large programs. The method we propose for optimizing tests should then be scalable enough to be applied on large systems. As shown with VIRTUALMEETING case study, the approach can be applied on several thousand statements. Yet, on a much bigger system, even if it is fully automated, the optimization process may be really time-consuming.

To deal with scalability, we propose an incremental methodology, which reduces the diagnosis scope step-by-step. In a large system, a subset of code must be selected and test cases are improved to maximize the number of DBBs (i.e. minimize the size of DBBs) in this subset. Several techniques can be used to select this sub-set of code that contains the fault to locate:

- The tester expertise: She may allow selecting a particular sub-set of the system's classes regarding the test cases that failed. More generally, the tester may know from which sub-system the problem comes.
- Using failed test cases execution traces: if an error is detected by a test, there is a faulty statement in its trace. To locate this faulty statement, the test suite can be optimized to distinguish as many DBBs as possible in the set of statements executed by the test case that detects the error.
- Using a diagnosis algorithm: the most suspicious DBBs are automatically selected by the diagnosis algorithm. Then, the local optimization process is used to split those DBBs.

Then, the test case optimization will try to break the DBBs in this subset of the program code. Figure 8 summarizes this local test suite optimization process.

Using the local optimization process allows an incremental approach for fault localization and improves the scalability of the technique. Next section discusses some remaining problems regarding the link between the testing task and the diagnosis one.

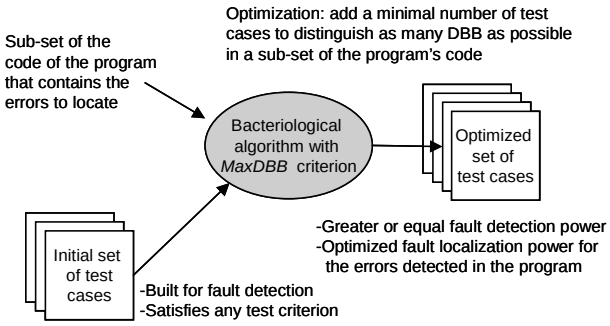


Figure 8 - Local test suite optimization process

6.2 Using assertions to break DBBs

The TfD criterion has been defined to maximize statement distinction while cross-checking the test cases traces. However, all statements cannot be individually distinguished: the initialization code is an example of large statements block in which the fault localization algorithm fails producing an accurate diagnosis. To solve the problem, some complementary information is needed (in addition to the test cases traces) as an input of the diagnosis algorithm. It must allow distinguishing statements that belong to a same static basic block. Adding assertions in the code of the program under test may be a way to split the static blocks. For instance, the Design by Contract approach [12], which consists in adding pre and post conditions to every method of the program, may be used.

7. CONCLUSION AND FUTURE WORK

This work establishes an explicit connection between testing and diagnosis. Specifically, the main contribution is the identification of a "test for diagnosis criterion" which is defined to ensure a satisfactory "fault locating power" for a test suite with respect to the studied diagnosis techniques. This criterion consists in maximizing the number of distinguished dynamic basic blocks. The benefits of such a criterion are two-fold:

- It allows minimizing the number of test cases required for an accurate diagnosis. It thus reconciles the actual test practices with the diagnosis requirements.
- It provides a way to automatically estimate the quality of a particular test suite with respect to the diagnosis requirements. This estimation can be used to improve a test suite in order to improve the efficiency of the diagnosis aid technique.

The technique is experimentally validated on an OO system made of over a thousand statements. We detail a method to apply this technique on large programs. In future work, experiments will be carried out to evaluate whether classical test adequacy criteria can efficiently isolate DBBs.

8. ACKNOWLEDGEMENTS

The authors are grateful to Sebastian Elbaum, Jim Jones and the anonymous reviewers for their helpful comments on the preliminary version of the paper.

9. REFERENCES

[1] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault Localization using Execution Slices and Dataflow Tests. Proceedings of ISSRE'95 (Int. Symposium on Software Reliability Engineering), Toulouse, France, October 1995.

[2] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Automatic Test Cases Optimization using a Bacteriological Adaptation Model: Application to .NET Components. Proceedings of ASE'02 (Automated Software Engineering), Edimburgh, Scotland, UK, September 2002.

[3] B. Baudry, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel. An Original Approach for Automatic Test Cases Optimization: a Bacteriologic Algorithm. IEEE Software, 2005. 22(2): 76-82.

[4] H. Cleve and A. Zeller. Locating Causes of Program Failures. Proceedings of ICSE (International Conference on Software Engineering), St. Louis, Missouri, USA, May 2005.

[5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight Defect Localization for Java. Proceedings of ECOOP'05 (European Conference on Object-Oriented Programming), Glasgow, Scotland, July 2005.

[6] R. DeMillo, R. Lipton, and F. Sayward. Hints on Test Data Selection : Help For The Practicing Programmer. IEEE Computer, 1978. 11(4): 34 - 41.

[7] F. Fleurey, B. Baudry, and Y. Le Traon. Improving Test Cases for Accurate Diagnosis. Accessed on: May 2005. http://www.irisa.fr/triskell/results/Diagnosis/index.htm

[8] J.A. Jones and M.J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault Localization Technique. Proceedings of ASE'05 (Automated Software Engineering), Long Beach, California, USA, November 2005.

[9] J.A. Jones, M.J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. Proceedings of ICSE'02 (Int. Conference in Software Engineering), Orlando, FL, USA, May 2002.

[10] M. Khalil, Y. Le Traon, and C. Robach. Towards an automatic diagnosis for high-level design validation. Proceedings of International Test Conference, Washington, DC, USA, October 1998.

[11] B. Korel. Dynamic method for software test data generation. Software Testing, Verification and Reliability, 1992. 2(4): 203 - 213.

[12] B. Meyer. Object-oriented software construction. Prentice Hall, 1992.

[13] C.C. Michael, G. McGraw, and M.A. Schatz. Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, 2001. 27(12): 1085 - 1110.

[14] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. ACM Transactions on Software Engineering and Methodology, 1996. 5(2): 99 - 118.

[15] M.L. Rosenzweig. Species Diversity In Space and Time. Cambridge University Press, 1995.

[16] J. Wegener, A. Baresel, and H. Stahmer. Evolutionary Test Environment for Automatic Structural Testing. Information and Software Technology, 2001. 43(14): 841 - 854.

[17] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering, 2002. 28(2): 183-200.